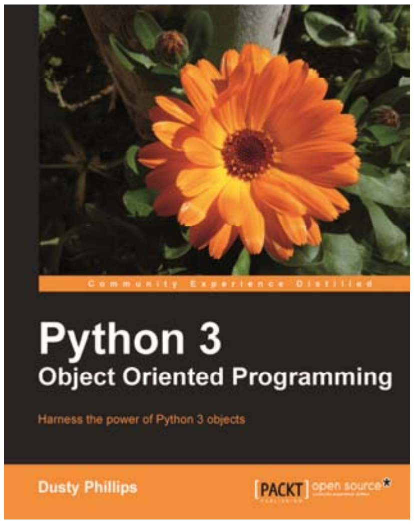
Python 3 Object Oriented Programming

Dusty Phillips



Chapter No.7

"Python Object-oriented Shortcuts"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.7 "Python Object-oriented Shortcuts"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Dusty Phillips is a Canadian freelance software developer, teacher, martial artist, and open source aficionado. He is closely affiliated with the Arch Linux community and other open source projects. He maintains the Arch Linux storefronts, and compiled the popular Arch Linux Handbook. Dusty holds a Master's degree in Computer Science specializing in Human-Computer Interaction. He currently has six different Python interpreters installed on his computer.

I would like to thank my editors, Steven Wilding and Mayuri Kokate for well-timed encouragement and feedback. Many thanks to friend and mentor Jason Chu for getting me started in Python and for patiently answering numerous questions on Python, GIT, and life over the years. Thanks to my father, C. C. Phillips, for inspiring me to write while editing his terrific works of fiction. Finally, thanks to every person who has said they can't wait to buy my book; your enthusiasm has been a huge motivational force.

Python 3 Object Oriented Programming

This book will introduce you to the terminology of the object-oriented paradigm, focusing on object-oriented design with step-by-step examples. It will take you from simple inheritance, one of the most useful tools in the object-oriented programmer's toolbox, all the way through to cooperative inheritance, one of the most complicated. You will be able to raise, handle, define, and manipulate exceptions.

You will be able to integrate the object-oriented and not-so-object-oriented aspects of Python. You will also be able to create maintainable applications by studying higherlevel design patterns. You'll learn the complexities of string and fi le manipulation and how Python distinguishes between binary and textual data. Not one, but two very powerful automated testing systems will be introduced to you. You'll understand the joy of unit testing and just how easy unit tests are to create. You'll even study higher-level libraries such as database connectors and GUI toolkits and how they apply object-oriented principles.

What This Book Covers

Chapter 1, Object-oriented Design covers important object-oriented concepts. It deals mainly with abstraction, classes, encapsulation, and inheritance. We also briefly look into UML to model our classes and objects.

Chapter 2, Objects in Python discusses classes and objects and how they are used in Python. We will learn about attributes and behaviors in Python objects, and also the organization of classes into packages and modules. And lastly we shall see how to protect our data.

Chapter 3, When Objects are Alike gives us a more in-depth look into inheritance. It covers multiple inheritance and shows us how to inherit from built-ins. This chapter also covers polymorphism and duck typing.

Chapter 4, Expecting the Unexpected looks into exceptions and exception handling. We shall learn how to create our own exceptions. It also deals with the use of exceptions for program flow control.

Chapter 5, When to Use Object-oriented Programming deals with objects; when to create and use them. We will see how to wrap data using properties, and restricting data access. This chapter also discusses the DRY principle and how not to repeat code.

Chapter 6, Python Data Structures covers object-oriented features of data structures. This chapter mainly deals with tuples, dictionaries, lists, and sets. We will also see how to extend built-in objects.

Chapter 7, Python Object-oriented Shortcuts as the name suggests, deals with little time-savers in Python. We shall look at many useful built-in functions, then move on to using comprehensions in lists, sets, and dictionaries. We will learn about generators, method overloading, and default arguments. We shall also see how to use functions as objects.

Chapter 8, Python Design Patterns I first introduces us to Python design patterns. We shall then see the decorator pattern, observer pattern, strategy pattern, state pattern, singleton pattern, and template pattern. These patterns are discussed with suitable examples and programs implemented in Python.

Chapter 9, Python Design Patterns II picks up where the previous chapter left us. We shall see the adapter pattern, facade pattern, flyweight pattern, command pattern, abstract pattern, and composite pattern with suitable examples in Python.

Chapter 10, Files and Strings looks at strings and string formatting. Bytes and byte arrays are also discussed. We shall also look at fi les, and how to write and read data to and from fi les. We shall look at ways to store and pickle objects, and finally the chapter discusses serializing objects.

Chapter 11, Testing Object-oriented Programs opens with the use of testing and why testing is so important. It focuses on test-driven development. We shall see how to use the `unittest` module, and also the `py.test` automated testing suite. Lastly we shall cover code coverage using `coverage.py`.

Chapter 12, Common Python 3 Libraries concentrates on libraries and their utilization in application building. We shall build databases using SQLAlchemy, and user interfaces TkInter and PyQt. The chapter goes on to discuss how to construct XML documents and we shall see how to use ElementTree and lxml. Lastly we will use CherryPy and Jinja to create a web application.

7 Python Object-oriented Shortcuts

Now let's look at some aspects of Python that appear more reminiscent of structural or functional programming than object-oriented programming. Although object-oriented programming is the most popular kid on the block these days, the old paradigms still offer useful tools. Most of these tools are really syntactic sugar over an underlying object-oriented implementation; we can think of them as a further abstraction layer built on top of the (already abstracted) object-oriented paradigm. In this chapter we'll be covering:

- Built-in functions that take care of common tasks in one call
- List, set, and dictionary comprehensions
- Generators
- An alternative to method overloading
- Functions as objects

Python built-in functions

There are numerous functions in Python that perform a task or calculate a result on certain objects without being methods on the class. Their purpose is to abstract common calculations that apply to many types of classes. This is applied duck typing; these functions accept objects with certain attributes or methods that satisfy a given interface, and are able to perform generic tasks on the object.

Len

The simplest example is the `len()` function. This function counts the number of items in some kind of container object such as a dictionary or list. For example:

```
>>> len([1,2,3,4])
4
```

Why don't these objects have a length property instead of having to call a function on them? Technically, they do. Most objects that `len()` will apply to have a method called `__len__()` that returns the same value. So `len(myobj)` seems to call `myobj.__len__()`.

Why should we use the function instead of the method? Obviously the method is a special method with double-underscores suggesting that we shouldn't call it directly. There must be an explanation for this. The Python developers don't make such design decisions lightly.

The main reason is efficiency. When we call `__len__` on an object, the object has to look the method up in its namespace, and, if the special `__getattribute__` method (which is called every time an attribute or method on an object is accessed) is defined on that object, it has to be called as well. Further `__getattribute__` for that particular method may have been written to do something nasty like refusing to give us access to special methods such as `__len__`! The `len` function doesn't encounter any of this. It actually calls the `__len__` function on the underlying class, so `len(myobj)` maps to `MyObj.__len__(myobj)`.

Another reason is maintainability. In the future, the Python developers may want to change `len()` so that it can calculate the length of objects that don't have a `__len__`, for example by counting the number of items returned in an iterator. They'll only have to change one function instead of countless `__len__` methods across the board.

Reversed

The `reversed()` function takes any sequence as input, and returns a copy of that sequence in reverse order. It is normally used in `for` loops when we want to loop over items from back to front.

Similar to `len`, `reversed` calls the `__reversed__()` function on the **class** for the parameter. If that method does not exist, `reversed` builds the reversed sequence itself using calls to `__len__` and `__getitem__`. We only need to override `__reversed__` if we want to somehow customize or optimize the process:

```
normal_list=[1,2,3,4,5]

class CustomSequence():
    def __len__(self):
        return 5

    def __getitem__(self, index):
        return "x{0}".format(index)

class FunkyBackwards(CustomSequence):
    def __reversed__(self):
        return "BACKWARDS!"

for seq in normal_list, CustomSequence(), FunkyBackwards():
    print("\n{}: ".format(seq.__class__.__name__), end="")
    for item in reversed(seq):
        print(item, end=", ")
```

The `for` loops at the end print the reversed versions of a normal list, and instances of the two custom sequences. The output shows that `reversed` works on all three of them, but has very different results when we define `__reversed__` ourselves:

```
list: 5, 4, 3, 2, 1,
CustomSequence: x4, x3, x2, x1, x0,
FunkyBackwards: B, A, C, K, W, A, R, D, S, !,
```

Note: the above two classes aren't very good sequences, as they don't define a proper version of `__iter__` so a forward `for` loop over them will never end.

Enumerate

Sometimes when we're looping over an iterable object in a `for` loop, we want access to the index (the current position in the list) of the current item being processed. The `for` loop doesn't provide us with indexes, but the `enumerate` function gives us something better: it creates a list of tuples, where the first object in each tuple is the index and the second is the original item.

This is useful if we want to use index numbers directly. Consider some simple code that outputs all the lines in a file with line numbers:

```
import sys
filename = sys.argv[1]

with open(filename) as file:
    for index, line in enumerate(file):
        print("{0}: {1}".format(index+1, line), end='')
```

Running this code on itself as the input file shows how it works:

```
1: import sys
2: filename = sys.argv[1]
3:
4: with open(filename) as file:
5:     for index, line in enumerate(file):
6:         print("{0}: {1}".format(index+1, line), end='')
```

The `enumerate` function returns a list of tuples, our `for` loop splits each tuple into two values, and the `print` statement formats them together. It adds one to the index for each line number, since `enumerate`, like all sequences is zero based.

Zip

The `zip` function is one of the least object-oriented functions in Python's collection. It takes two or more sequences and creates a new sequence of tuples. Each tuple contains one element from each list.

This is easily explained by an example; let's look at parsing a text file. Text data is often stored in tab-delimited format, with a "header" row as the first line in the file, and each line below it describing data for a unique record. A simple contact list in tab-delimited format might look like this:

```
first     last   email
john      smith  jsmith@example.com
jane      doan   janed@example.com
david     neilson      dn@example.com
```

A simple parser for this file can use `zip` to create lists of tuples that map headers to values. These lists can be used to create a dictionary, a much easier object to work with in Python than a file!

```
import sys
filename = sys.argv[1]
```

```
contacts = []
with open(filename) as file:
    header = file.readline().strip().split('\t')
    for line in file:
        line = line.strip().split('\t')
        contact_map = zip(header, line)
        contacts.append(dict(contact_map))

for contact in contacts:
    print("email: {email} -- {last}, {first}".format(
        **contact))
```

What's actually happening here? First we open the file, whose name is provided on the command line, and read the first line. We strip the trailing newline, and split what's left into a list of three elements. We pass `'\t'` into the strip method to indicate that the string should be split at tab characters. The resulting header list looks like `["first", "last", "email"]`.

Next, we loop over the remaining lines in the file (after the header). We split each line into three elements. Then, we use `zip` to create a sequence of tuples for each line. The first sequence would look like `[("first", "john"), ("last", "smith"), ("email", "jsmith@example.com")]`.

Pay attention to what `zip` is doing. The first list contains headers; the second contains values. The `zip` function created a tuple of header/value pairs for each matchup.

The `dict` constructor takes the list of tuples, and maps the first element to a key and the second to a value to create a dictionary. The result is added to a list.

At this point, we are free to use dictionaries to do all sorts of contact-related activities. For testing, we simply loop over the contacts and output them in a different format. The format line, as usual, takes variable arguments and keyword arguments. The use of `**contact` automatically converts the dictionary to a bunch of keyword arguments (we'll understand this syntax before the end of the chapter) Here's the output:

```
email: jsmith@example.com -- smith, john
email: janed@example.com -- doan, jane
email: dn@example.com -- neilson, david
```

If we provide `zip` with lists of different lengths, it will stop at the end of the shortest list. There aren't many useful applications of this feature, but `zip` will not raise an exception if that is the case. We can always check the list lengths and add empty values to the shorter list, if necessary.

The zip function is actually the inverse of itself. It can take multiple sequences and combine them into a single sequence of tuples. Because tuples are also sequences, we can "unzip" a zipped list of tuples by zipping it again. Huh? Have a look at this example:

```
>>> list_one = ['a', 'b', 'c']
>>> list_two = [1, 2, 3]
>>> zipped = zip(list_one, list_two)
>>> zipped = list(zipped)
>>> zipped
[('a', 1), ('b', 2), ('c', 3)]
>>> unzipped = zip(*zipped)
>>> list(unzipped)
[('a', 'b', 'c'), (1, 2, 3)]
```

First we zip the two lists and convert the result into a list of tuples. We can then use parameter unpacking to pass these individual sequences as arguments to the zip function. zip matches the first value in each tuple into one sequence and the second value into a second sequence; the result is the same two sequences we started with!

Other functions

Another key function is sorted(), which takes an iterable as input, and returns a list of the items in sorted order. It is very similar to the sort() method on lists, the difference being that it works on all iterables, not just lists.

Like list.sort, sorted accepts a key argument that allows us to provide a function to return a sort value for each input. It can also accept a reverse argument.

Three more functions that operate on sequences are min, max, and sum. These each take a sequence as input, and return the minimum or maximum value, or the sum of all values in the sequence. Naturally, sum only works if all values in the sequence are numbers. The max and min functions use the same kind of comparison mechanism as sorted and list.sort, and allow us to define a similar key function. For example, the following code uses enumerate, max, and min to return the indices of the values in a list with the maximum and minimum value:

```
def min_max_indexes(seq):
    minimum = min(enumerate(seq), key=lambda s: s[1])
    maximum = max(enumerate(seq), key=lambda s: s[1])
    return minimum[0], maximum[0]
```

The `enumerate` call converts the sequence into `(index, item)` tuples. The `lambda` function passed in as a `key` tells the function to search the second item in each tuple (the original item). The `minimum` and `maximum` variables are then set to the appropriate tuples returned by `enumerate`. The `return` statement takes the first value (the index from enumerate) of each tuple and returns the pair. The following interactive session shows how the returned values are, indeed, the indices of the minimum and maximum values:

```
>>> alist = [5,0,1,4,6,3]
>>> min_max_indexes(alist)
(1, 4)
>>> alist[1], alist[4]
(0, 6)
```

We've only touched on a few of the more important Python built-in functions. There are numerous others in the standard library, including:

- `all` and `any`, which accept an iterable and returns `True` if all, or any, of the items evaluate to true (that is a non-empty string or list, a non-zero number, an object that is not `None`, or the literal `True`).
- `eval`, `exec`, and `compile`, which execute string as code inside the interpreter.
- `hasattr`, `getattr`, `setattr`, and `delattr`, which allow attributes on an object to be manipulated as string names.
- And many more! See the interpreter help documentation for each of the functions listed in `dir(__builtins__)`.

Comprehensions

We've already seen a lot of Python's `for` loop. It allows us to loop over any object that supports the iterable protocol and do something specific with each of the elements in turn.

Supporting the iterable protocol simply means an object has an `__iter__` method that returns another object that supports the iterator protocol. Supporting the iterator protocol is a fancy way of saying it has a `__next__` method that either returns the next object in the sequence, or raises a `StopIteration` exception when all objects have been returned.

As you can see, the `for` statement, in spite of not looking terribly object-oriented, is actually a shortcut to some extremely object-oriented designs. Keep this in mind as we discuss comprehensions, as they, too, appear to be the polar opposite of an object-oriented tool. Yet, they use the same iteration protocol as `for` loops. They're just another kind of shortcut.

List comprehensions

List comprehensions are one of the most powerful tools in Python, so people tend to think of them as advanced. They're not. Indeed, I've taken the liberty of littering previous examples with comprehensions and assuming you'd understand them. While it's true that advanced programmers use comprehensions a lot, it's not because they're advanced, it's because they're trivial, and handle some of the most common operations in programming.

Let's have a look at one of those common operations, namely, converting a list of items into a list of related items. Specifically, let's assume we just read a list of strings from a file, and now we want to convert it to a list of integers; we know every item in the list is an integer, and we want to do some activity (say, calculate an average) on those numbers. Here's one simple way to approach it:

```
input_strings = ['1', '5', '28', '131', '3']

output_integers = []
for num in input_strings:
    output_integers.append(int(num))
```

This works fine, it's only three lines of code. If you aren't used to comprehensions, you may not even think it looks ugly! Now, look at the same code using a list comprehension:

```
input_strings = ['1', '5', '28', '131', '3']

output_integers = [int(num) for num in input_strings]
```

We're down to one line and we've dropped an `append` method call. Overall, it's pretty easy to tell what's going on, even if you're not used to comprehension syntax.

The square brackets show we're creating a list. Inside this list is a `for` loop that loops over each item in the input sequence. The only thing that may be confusing is what's happening between the list's opening brace and the start of the `for` loop. Whatever happens here is applied to **each** of the items in the input list. The item in question is referenced by the `num` variable from the loop. So it's converting each such item to an `int`.

That's all there is to a basic list comprehension. They are not so advanced, after all! Comprehensions are highly optimized; list comprehensions are far faster than `for` loops when we are looping over a huge number of items. If readability alone isn't a convincing reason to use them as much as possible, then speed should be.

Converting one list of items into a related list isn't the only thing we can do with a list comprehension. We can also choose to exclude certain values by adding an `if` statement inside the comprehension. Have a look:

```
output_ints = [int(n) for n in input_strings if len(n) < 3]
```

I shortened the name of the variable from `num` to `n` and the result variable to `output_ints` so it would still fit on one line. Other than that, all that's different between this example and the previous one is the `if len(n) < 3` part. This extra code excludes any strings with more than two characters. The `if` statement is applied before the `int` function, so it's testing the length of a string. Since our input strings are all integers at heart, it refers to any number over ninety-nine. Now **that** is all there is to list comprehensions! We use them to map input values to output values, applying a filter along the way to exclude any values that don't meet a specific condition.

Any iterable can be the input to a list comprehension; anything we can wrap in a `for` loop can also be placed inside a comprehension. For example, text files are iterable; each call to `__next__` on the file's iterator will return one line of the file. The contact file example we used earlier (to try out the `zip` function) can use a list comprehension instead:

```
import sys
filename = sys.argv[1]

with open(filename) as file:
    header = file.readline().strip().split('\t')
    contacts = [
            dict(
                zip(header, line.strip().split('\t'))
                ) for line in file
            ]
```

This time, I've added some whitespace to make it more readable (list comprehensions don't **have** to fit on one line). This example is doing the same thing as the previous version: creating a list of dictionaries from the zipped header and split lines for each line in the file.

Er, what? Don't worry if that code or explanation doesn't make sense; it's a bit confusing. One little list comprehension is doing a pile of work here, and the code is hard to understand, read, and ultimately, maintain. This example shows that list comprehensions aren't always the best solution; most programmers would agree that the earlier `for` loop is more readable than this version. Remember: the tools we are provided with should not be abused! Always pick the right tool for the job, and that job is writing maintainable code.

Set and dictionary comprehensions

Comprehensions aren't restricted to lists. We can use a similar syntax with braces to create sets and dictionaries as well. Let's start with sets. One way to create a set is to wrap a list comprehension in the `set()` constructor, which converts it to a set. But why waste memory on an intermediate list that gets discarded when we can create a set directly?

Here's an example that uses a named tuple to model author/title/genre triads, and then retrieves a set of all the authors that write in a specific genre:

```
from collections import namedtuple

Book = namedtuple("Book", "author title genre")
books = [
        Book("Pratchett", "Nightwatch", "fantasy"),
        Book("Pratchett", "Thief Of Time", "fantasy"),
        Book("Le Guin", "The Dispossessed", "scifi"),
        Book("Le Guin", "A Wizard Of Earthsea", "fantasy"),
        Book("Turner", "The Thief", "fantasy"),
        Book("Phillips", "Preston Diamond", "western"),
        Book("Phillips", "Twice Upon A Time", "scifi"),
        ]

fantasy_authors = {
        b.author for b in books if b.genre == 'fantasy'}
```

That set comprehension sure is short in comparison to the set up required! If we'd used a list comprehension, of course, Terry Pratchett would have been listed twice. As it is, the nature of sets removes the duplicates and we end up with:

```
>>> fantasy_authors
{'Turner', 'Pratchett', 'Le Guin'}
```

We can introduce a colon to create a dictionary comprehension. This converts a sequence into a dictionary using `key : value` pairs. For example, it may be useful to quickly look up the author or genre in a dictionary if we know the title. We can use a dictionary comprehension to map titles to book objects:

```
fantasy_titles = {
        b.title: b for b in books if b.genre == 'fantasy'}
```

Now we have a dictionary and can look up books by title using the normal syntax.

In summary, comprehensions are not advanced Python, and they aren't "non-object-oriented" tools that should be avoided. They are simply a more concise and optimized syntax for creating a list, set, or dictionary from an existing sequence.

Generator expressions

Sometimes we want to process a new sequence without placing a new list, set, or dictionary into system memory. If we're just looping over items one at a time, and don't actually care about having a final container object created, creating that container is a waste of memory. When processing one item at a time, we only need the current object stored in memory at any one moment. But when we create a container, all the objects have to be stored in that container before we start processing them.

For example, consider a program that processes log files. A very simple log might contain information in this format:

```
Jan 26, 2010 11:25:25   DEBUG         This is a debugging message.
Jan 26, 2010 11:25:36   INFO          This is an information method.
Jan 26, 2010 11:25:46   WARNING       This is a warning. It could be
serious.
Jan 26, 2010 11:25:52   WARNING       Another warning sent.
Jan 26, 2010 11:25:59   INFO          Here's some information.
Jan 26, 2010 11:26:13   DEBUG         Debug messages are only useful if
you want to figure something out.
Jan 26, 2010 11:26:32   INFO          Information is usually harmless,
but helpful.
Jan 26, 2010 11:26:40   WARNING       Warnings should be heeded.
Jan 26, 2010 11:26:54   WARNING       Watch for warnings.
```

Log files for popular web servers, databases, or e-mail servers can contain many gigabytes of data. If we want to process each line in the log, we don't want to use a list comprehension on those lines; it would create a list containing every line in the file. This probably wouldn't fit in memory and could bring the computer to its knees, depending on the operating system.

If we used a `for` loop on the log file, we could process one line at a time before reading the next one into memory. Wouldn't be nice if we could use comprehension syntax to get the same effect?

This is where generator expressions come in. They use the same syntax as comprehensions, but they don't create a final container object. To create a generator expression, wrap the comprehension in `()` instead of `[]` or `{}`.

The following code parses a log file in the previously presented format, and outputs a new log file that contains only the `WARNING` lines:

```
import sys

inname = sys.argv[1]
outname = sys.argv[2]

with open(inname) as infile:
    with open(outname, "w") as outfile:
        warnings = (l for l in infile if 'WARNING' in l)
        for l in warnings:
            outfile.write(l)
```

This program takes the two filenames on the command line, uses a generator expression to filter out the warnings (in this case, it uses the `if` syntax, and leaves the line unmodified), and then outputs the warnings to another file. If we run it on our sample file, the output looks like this:

```
Jan 26, 2010 11:25:46   WARNING      This is a warning. It could be
serious.
Jan 26, 2010 11:25:52   WARNING      Another warning sent.
Jan 26, 2010 11:26:40   WARNING      Warnings should be heeded.
Jan 26, 2010 11:26:54   WARNING      Watch for warnings.
```

Of course, with such a short input file, we could have safely used a list comprehension, but if the file is millions of lines long, the generator expression will have a huge impact on both memory and speed.

Generator expressions can also be useful inside function calls. For example, we can call `sum`, `min`, or `max` on a generator expression instead of a list, since these functions process one object at a time. We're only interested in the result, not any intermediate container.

In general, a generator expression should be used whenever possible. If we don't actually need a list, set, or dictionary, but simply need to filter or convert items in a sequence, a generator expression will be most efficient. If we need to know the length of a list, or sort the result, remove duplicates, or create a dictionary, we'll have to use the comprehension syntax.

Generators

Generator expressions are actually a sort of comprehension too; they compress the more advanced (this time it really is more advanced!) generator syntax into one line. The greater generator syntax looks even less object-oriented than anything we've seen, but we'll discover that once again, it is a simple syntax shortcut to create a kind of object.

Let's take the log file example a little further. If we want to delete the `WARNING` column from our output file (since it's redundant; this file contains only warnings), we have several options, at various levels of readability. We can do it with a generator expression:

```
import sys
inname, outname = sys.argv[1:3]

with open(inname) as infile:
    with open(outname, "w") as outfile:
        warnings = (l.replace('\tWARNING', '')
                for l in infile if 'WARNING' in l)
        for l in warnings:
            outfile.write(l)
```

That's perfectly readable, though I wouldn't want to make the expression any more complicated than that. We could also do it with a normal `for` loop:

```
import sys
inname, outname = sys.argv[1:3]

with open(inname) as infile:
    with open(outname, "w") as outfile:
        for l in infile:
            if 'WARNING' in l:
                outfile.write(l.replace('\tWARNING', ''))
```

That's maintainable, but so many levels of indent in so few lines is kind of ugly. Now let's consider a truly object-oriented solution, without any shortcuts:

```
import sys
inname, outname = sys.argv[1:3]

class WarningFilter:
    def __init__(self, insequence):
        self.insequence = insequence
    def __iter__(self):
        return self
    def __next__(self):
        l = self.insequence.readline()
        while l and 'WARNING' not in l:
            l = self.insequence.readline()
        if not l:
            raise StopIteration
        return l.replace('\tWARNING', '')

with open(inname) as infile:
    with open(outname, "w") as outfile:
        filter = WarningFilter(infile)
        for l in filter:
            outfile.write(l)
```

No doubt about it: that is ugly and difficult to read. What is happening here? Well, we created an object that takes a file object as input, and then provides a `__next__` method to allow it to work as an iterator in `for` loops. That method reads lines from the file, discarding them if they are not `WARNING` lines. When it encounters a `WARNING` line, it returns it, and the `for` loop will call `__next__` again to get the next line. When we run out of lines, we raise `StopIteration` to tell the loop we're finished. It's pretty ugly compared to the other examples, but it's also powerful; now that we have a class in our hands, we can do whatever we want to it.

With that background behind us, we finally get to see generators in action. This next example does exactly the same thing as the previous one: it creates an object that allows us to loop over the input:

```
import sys
inname, outname = sys.argv[1:3]

def warnings_filter(insequence):
    for l in insequence:
        if 'WARNING' in l:
            yield l.replace('\tWARNING', '')

with open(inname) as infile:
    with open(outname, "w") as outfile:
        filter = warnings_filter(infile)
        for l in filter:
            outfile.write(l)
```

OK, that's pretty readable, maybe... at least it's short. But what on earth is going on here, it doesn't make sense. And what is `yield`, anyway?

Last question first: `yield` is the key to generators. When Python sees `yield` in a function, it takes that function and wraps it up in an object not unlike the one in our previous example. Think of the `yield` statement as similar to the `return` statement; it exits the function and returns a line. Unlike `return`, when the function is called again, it will start where it left off; on the line after the `yield` statement. In this example, there is no line after the `yield` statement, so it jumps to the next iteration of the `for` loop. Since the `yield` statement is inside an `if` statement, it only yields lines that contain `WARNING`.

While it looks like that function is simply looping over the lines, it is really creating an object; a generator object:

```
>>> print(warnings_filter([]))
<generator object warnings_filter at 0xb728c6bc>
```

I passed an empty list into the function to act as an iterator. All the function does is create and return a generator object. That object has `__iter__` and `__next__` methods on it, much like the one we created in the previous example. Whenever `__next__` is called, the generator runs the function until it finds a `yield` statement. It then returns the value from `yield`, and the next time `__next__` is called, it picks up where it left off.

This use of generators isn't that advanced, but if you don't realize the function is creating an object, it can seem magical. We can even have multiple calls to `yield` in a single function; it will simply pick up at the most recent `yield` and continue to the next one.

There is even more to generators than what we have covered. We can send values back into generators when calling `yield`, turning them into a dark art called coroutines. While technically objects, coroutines encourage us to think very differently from the object-oriented principles we've been discussing, and are beyond the scope of this book. Do a search if you are interested in learning more about them.

An alternative to method overloading

One prominent feature of many object-oriented programming languages is a tool called **method overloading**. Method overloading simply refers to having multiple methods with the same name that accept different sets of arguments. In statically typed languages, this is useful if we want to have a method that accepts either an integer or a string, for example. In non-object-oriented languages we might need two functions called `add_s` and `add_i` to accommodate such situations. In statically typed object-oriented languages, we'd need two methods, both called `add`, one that accepts strings, and one that accepts integers.

In Python, we only need one method, which accepts any type of object. It may have to do some testing on the object type (for example, if it is a string, convert it to an integer), but only one method is required.

However, method overloading is also useful when we want a method with the same name to accept different numbers or sets of arguments. For example, an e-mail message method might come in two versions, one of which accepts an argument for the from e-mail address. The other method might look up a default from address instead. Python doesn't permit multiple methods with the same name, but it does provide a different, equally flexible, interface.

We've seen some of the possible ways to send arguments to methods and functions in previous examples, but now we'll cover all the details. The simplest function accepts no arguments. We probably don't need an example, but here's one for completeness:

```
def no_args():
    pass
```

and here's how it's called:

```
no_args()
```

A function that does accept arguments will provide the names of those arguments in a comma-separated list. Only the name of each argument needs to be supplied.

When calling the function, these **positional** arguments must be specified in order, and none can be missed or skipped. This is the most common way we've specified arguments in our previous examples:

```
def mandatory_args(x, y, z):
    pass
```

and to call it:

```
mandatory_args("a string", a_variable, 5)
```

Any type of object can be passed as an argument: an object, a container, a primitive, even functions and classes. The above call shows a hard-coded string, an unknown variable, and an integer passed into the function.

Default arguments

If we want to make an argument optional, rather than creating a second method with a different set of arguments, we can specify a default value in a single method, using an equals sign. If the calling code does not supply this argument, it will be assigned a default value. However, the calling code can still choose to override the default by passing in a different value. Often, a default value of `None`, or an empty string or list is suitable.

Here's a function definition with default arguments:

```
def default_arguments(x, y, z, a="Some String", b=False):
    pass
```

The first three arguments are still mandatory and must be passed by the calling code. The last two parameters have default arguments supplied.

There are several ways we can call this function. We can supply all arguments in order, as though all the arguments were positional arguments.

```
kwargs("a string", variable, 8, "", True)
```

Or we can supply just the mandatory arguments in order, leaving the keyword arguments to be assigned their default values:

```
kwargs("a longer string", some_variable, 14)
```

We can also use the equals sign syntax when calling a function to provide values in a different order or to skip default values that we aren't interested in. For example, we can skip the first keyword arguments and supply the second one:

```
kwargs("a string", variable, 14, b=True)
```

Surprisingly, we can even use the equals sign syntax to mix up the order of positional arguments, so long as all of them are supplied.

```
>>> kwargs(y=1,z=2,x=3,a="hi")
3 1 2 hi False
```

With so many options, it may seem hard to pick one, but if you think of the positional arguments as an ordered list, and keyword arguments as sort of like a dictionary, you'll find that the correct layout tends to fall into place. If you need to require the caller to specify an argument, make it mandatory; if you have a sensible default, then make it a keyword argument. Choosing how to call the method normally takes care of itself, depending on which values need to be supplied, and which can be left at their defaults.

One thing to take note of with keyword arguments is that anything we provide as a default argument is evaluated when the function is first interpreted, not when it is called. This means we can't have dynamically generated default values. For example, the following code won't behave quite as expected:

```
number = 5
def funky_function(number=number):
    print(number)

number=6
funky_function(8)
funky_function()
print(number)
```

If we run this code, it outputs the number 8, first, but then it outputs the number 5 for the call with no arguments. We had set the variable to the number 6, as evidenced by the last line of output, but when the function is called, the number 5 is printed; the default value was calculated when the function was defined, not when it was called.

This is tricky with empty containers. For example, it is common to ask calling code to supply a list that our function is going to manipulate, but the list is optional. We'd like to make an empty list as a default argument. We can't do this; it will create only one list, when the code is first constructed:

```
>>> def hello(b=[]):
...     b.append('a')
...     print(b)
...
>>> hello()
['a']
>>> hello()
['a', 'a']
```

Whoops, that's not quite what we expected! The usual way to get around this is to make the default value `None`, and then use the idiom `if argument is None: arg = []` inside the method. Pay close attention!

Variable argument lists

Default values alone do not allow us all the flexible benefits of method overloading. The thing that makes Python really slick is the ability to write methods that accept an arbitrary number of positional or keyword arguments without explicitly naming them. We can also pass arbitrary lists and dictionaries into such functions.

For example, a function to accept a link or list of links and download the web pages could use such variadic arguments, or **varargs**. Instead of accepting a single value that is expected to be a list of links, we can accept an arbitrary number of arguments, where each argument is a different link. We do this by specifying the `*` operator in the function definition:

```
def get_pages(*links):
    for link in links:
        #download the link with urllib
        print(link)
```

The `*links` says "I'll accept any number of arguments and put them all in a list of strings named `links`". If we supply only one argument, it'll be a list with one element, if we supply no arguments, it'll be an empty list. Thus, all these function calls are valid:

```
get_pages()
get_pages('http://www.archlinux.org')
get_pages('http://www.archlinux.org',
        'http://ccphillips.net/')
```

We can also accept arbitrary keyword arguments. These arrive into the function as a dictionary. They are specified with two asterisks (as in `**kwargs`) in the function declaration. This tool is commonly used in configuration setups. The following class allows us to specify a set of options with default values:

```
class Options:
    default_options = {
            'port': 21,
            'host': 'localhost',
            'username': None,
            'password': None,
            'debug': False,
            }
    def __init__(self, **kwargs):
        self.options = dict(Options.default_options)
        self.options.update(kwargs)

    def __getitem__(self, key):
        return self.options[key]
```

All the interesting stuff in this class happens in the `__init__` method. We have a dictionary of default options and values at the class level. The first thing the `__init__` method does is make a copy of this dictionary. We do that instead of modifying the dictionary directly in case we instantiate two separate sets of options. (Remember, class level variables are shared between instances of the class.) Then, `__init__` uses the `update` method on the new dictionary to change any non-default values to those supplied as keyword arguments. The `__getitem__` method simply allows us to use the new class using indexing syntax. Here's a session demonstrating the class in action:

```
>>> options = Options(username="dusty", password="drowssap",
        debug=True)
>>> options['debug']
True
>>> options['port']
21
>>> options['username']
'dusty'
```

We're able to access our options instance using dictionary indexing syntax, and the dictionary includes both default values and the ones we set using keyword arguments.

The keyword argument syntax can be dangerous, as it may break the "explicit is better than implicit" rule. In the above example, it's possible to pass arbitrary keyword arguments to the `Options` initializer to represent options that don't exist in the default dictionary. This may not be a bad thing, depending on the purpose of the class, but it makes it hard for someone using the class to discover what valid options are available. It also makes it easy to enter a confusing typo ("Debug" instead of "debug", for example) that adds two options where only one should have existed.

The above example is not that bad if we instruct the user of the class to only pass default options (we could even add some code to enforce this rule). The options are documented in the class definition so it'll be easy to look them up.

Keyword arguments are also very useful when we need to accept arbitrary arguments to pass to a second function, but we don't know what those arguments will be. We saw this in action in *Chapter 3*, when we were building support for multiple inheritance.

We can, of course, combine the variable argument and variable keyword argument syntax in one function call, and we can use normal positional and default arguments as well. The following example is somewhat contrived, but demonstrates the four types in action:

```
import shutil
import os.path
def augmented_move(target_folder, *filenames,
        verbose=False, **specific):
    '''Move all filenames into the target_folder, allowing
    specific treatment of certain files.'''

    def print_verbose(message, filename):
        '''print the message only if verbose is enabled'''
        if verbose:
            print(message.format(filename))

    for filename in filenames:
        target_path = os.path.join(target_folder, filename)
        if filename in specific:
            if specific[filename] == 'ignore':
                print_verbose("Ignoring {0}", filename)
            elif specific[filename] == 'copy':
                print_verbose("Copying {0}", filename)
                shutil.copyfile(filename, target_path)
        else:
            print_verbose("Moving {0}", filename)
            shutil.move(filename, target_path)
```

This example will process an arbitrary list of files. The first argument is a target folder, and the default behavior is to move all remaining non-keyword argument files into that folder. Then there is a keyword-only argument, `verbose`, which tells us whether to print information on each file processed. Finally, we can supply a dictionary containing actions to perform on specific filenames; the default behavior is to move the file, but if a valid string action has been specified in the keyword arguments, it can be ignored or copied instead. Notice the ordering of the parameters in the function; first the positional argument is specified, then the `*filenames` list, then any specific keyword-only arguments, and finally, a `**specific` dictionary to hold remaining keyword arguments.

We create an inner helper function, `print_verbose`, which will print messages only if the `verbose` key has been set. This function keeps code readable by encapsulating this functionality into a single location.

In common cases, this function would likely be called as:

```
>>> augmented_move("move_here", "one", "two")
```

This command would move the files `one` and `two` into the `move_here` directory, assuming they exist (There's no error checking or exception handling in the function, so it would fail spectacularly if the files or target directory didn't exist). The move would occur without any output, since `verbose` is `False` by default.

If we want to see the output, we can call it with:

```
>>> augmented_move("move_here", "three", verbose=True)
Moving three
```

This moves one file, named `three`, and tells us what it's doing. Notice that it is impossible to specify `verbose` as a positional argument in this example; we **must** pass a keyword argument. Otherwise Python would think it was another filename in the `*filenames` list.

If we want to copy or ignore some of the files in the list, instead of moving them, we can pass additional keyword arguments:

```
>>> augmented_move("move_here", "four", "five", "six",
        four="copy", five="ignore")
```

This will move the sixth file and copy the fourth, but won't display any output, since we didn't specify `verbose`. Of course, we can do that, too, and keyword arguments can be supplied in any order:

```
>>> augmented_move("move_here", "seven", "eight", "nine",
        seven="copy", verbose=True, eight="ignore")
Copying seven
Ignoring eight
Moving nine
```

Unpacking arguments

There's one more nifty trick involving variable arguments and keyword arguments. We've used it in some of our previous examples, but it's never too late for an explanation. Given a list or dictionary of values, we can pass those values into a function as if they were normal positional or keyword arguments. Have a look at this code:

```
    def show_args(arg1, arg2, arg3="THREE"):
        print(arg1, arg2, arg3)
```

```
some_args = range(3)
more_args = {
        "arg1": "ONE",
        "arg2": "TWO"}

print("Unpacking a sequence:", end=" ")
show_args(*some_args)
print("Unpacking a dict:", end=" ")
show_args(**more_args)
```

Here's what it looks like when we run it:

```
Unpacking a sequence: 0 1 2
Unpacking a dict: ONE TWO THREE
```

The function accepts three arguments, one of which has a default value. But when we have a list of three arguments, we can use the `*` operator inside a function call to unpack it into the three arguments. If we have a dictionary of arguments, we can use the `**` syntax to unpack it as a collection of keyword arguments.

This is most often useful when mapping information that has been collected from user input or from an outside source (an internet page, a text file) to a function or method call.

Remember our earlier example that used headers and lines in a text file to create a list of dictionaries with contact information? Instead of just adding the dictionaries to a list, we could use keyword unpacking to pass the arguments to the `__init__` method on a specially built `Contact` object that accepts the same set of arguments. See if you can adapt the example to make this work.

Functions are objects too

Programming languages that over-emphasize object-oriented principles tend to frown on functions that are not methods. In such languages, you're expected to create an object to sort of wrap the single method involved. There are numerous situations where we'd like to pass around a small object that is simply called to perform an action. This is most frequently done in event-driven programming, such as graphical toolkits or asynchronous servers; we'll see some design patterns that use it in the next two chapters.

In Python, we don't need to wrap such methods in an object, because functions already are objects! We can set attributes on functions (though this isn't a common activity), and we can pass them around to be called at a later date. They even have a few special properties that can be accessed directly. Here's yet another contrived example:

```
def my_function():
    print("The Function Was Called")
my_function.description = "A silly function"

def second_function():
    print("The second was called")
second_function.description = "A sillier function."

def another_function(function):
    print("The description:", end=" ")
    print(function.description)
    print("The name:", end=" ")
    print(function.__name__)
    print("The class:", end=" ")
    print(function.__class__)
    print("Now I'll call the function passed in")
    function()

another_function(my_function)
another_function(second_function)
```

If we run this code, we can see that we were able to pass two different functions into our third function, and get different output for each one:

```
The description: A silly function
The name: my_function
The class: <class 'function'>
Now I'll call the function passed in
The Function Was Called
The description: A sillier function.
The name: second_function
The class: <class 'function'>
Now I'll call the function passed in
The second was called
```

We set an attribute on the function, named `description` (not very good descriptions, admittedly). We were also able to see the function's `__name__` attribute, and to access its class, demonstrating that the function really is an object with attributes. Then we called the function by using the callable syntax (the parentheses).

The fact that functions are top-level objects is most often used to pass them around to be executed at a later date, for example, when a certain condition has been satisfied. Let's build an event-driven timer that does just this:

```
import datetime
import time

class TimedEvent:
    def __init__(self, endtime, callback):
        self.endtime = endtime
        self.callback = callback

    def ready(self):
        return self.endtime <= datetime.datetime.now()

class Timer:
    def __init__(self):
        self.events = []

    def call_after(self, delay, callback):
        end_time = datetime.datetime.now() + \
                datetime.timedelta(seconds=delay)

        self.events.append(TimedEvent(end_time, callback))

    def run(self):
        while True:
            ready_events = (e for e in self.events if e.ready())
            for event in ready_events:
                event.callback(self)
                self.events.remove(event)
            time.sleep(0.5)
```

In production, this code should definitely have extra documentation using docstrings! The `call_after` method should at least mention that the `delay` is in seconds and that the `callback` function should accept one argument: the timer doing the calling.

We have two classes here. The `TimedEvent` class is not really meant to be accessed by other classes; all it does is store an `endtime` and `callback`. We could even use a `tuple` or `namedtuple` here, but as it is convenient to give the object a behavior that tells us whether or not the event is ready to run, we use a class instead.

The `Timer` class simply stores a list of upcoming events. It has a `call_after` method to add a new event. This method accepts a `delay` parameter representing the number of seconds to wait before executing the callback, and the `callback` itself: a function to be executed at the correct time. This callback function should accept one argument.

The `run` method is very simple; it uses a generator expression to filter out any events whose time has come, and executes them in order. The timer loop then continues indefinitely, so it has to be interrupted with a keyboard interrupt (*Ctrl* + *C* or *Ctrl* + *Break*). We sleep for half a second after each iteration so as to not grind the system to a halt.

The important things to note here are the lines that touch callback functions. The function is passed around like any other object and the timer never knows or cares what the original name of the function is or where it was defined. When it's time to call the function, the timer simply applies the parenthesis syntax to the stored variable.

Here's a set of callbacks that test the timer:

```
from timer import Timer
import datetime

def format_time(message, *args):
    now = datetime.datetime.now().strftime("%I:%M:%S")
    print(message.format(*args, now=now))

def one(timer):
    format_time("{now}: Called One")

def two(timer):
    format_time("{now}: Called Two")

def three(timer):
    format_time("{now}: Called Three")

class Repeater:
    def __init__(self):
        self.count = 0
    def repeater(self, timer):
        format_time("{now}: repeat {0}", self.count)
        timer.call_after(5, self.repeater)

timer = Timer()
timer.call_after(1, one)
timer.call_after(2, one)
timer.call_after(2, two)
timer.call_after(4, two)
timer.call_after(3, three)
timer.call_after(6, three)
repeater = Repeater()
timer.call_after(5, repeater.repeater)
format_time("{now}: Starting")
timer.run()
```

This example allows us to see how multiple callbacks interact with the timer. The first function is the `format_time` function. It uses the string `format` method to add the current time to the message, and illustrates variable arguments in action. The `format_time` method will accept any number of positional arguments, using variable argument syntax, which are then forwarded as positional arguments to the string's `format` method. After that we create three simple callback methods that simply output the current time and a short message telling us which callback has been fired.

The `Repeater` class demonstrates that methods can be used as callbacks too, since they are really just functions. It also shows why the `timer` argument to the callback functions is useful: we can add a new timed event to the timer from inside a presently running callback.

Then we simply create a timer and add several events to it that are called after different amounts of time. Then we start the timer running; the output shows that events are run in the expected order:

```
02:53:35: Starting
02:53:36: Called One
02:53:37: Called One
02:53:37: Called Two
02:53:38: Called Three
02:53:39: Called Two
02:53:40: repeat 0
02:53:41: Called Three
02:53:45: repeat 1
02:53:50: repeat 2
02:53:55: repeat 3
02:54:00: repeat 4
```

Using functions as attributes

One of the interesting effects of functions being objects is that they can be set as callable attributes on other objects. It is possible to add or change a function to an instantiated object:

```
class A:
    def print(self):
        print("my class is A")

def fake_print():
    print("my class is not A")

a = A()
a.print()
a.print = fake_print
a.print()
```

This code creates a very simple class with a `print` method that doesn't tell us anything we don't know. Then we create a new function that tells us something we don't believe.

When we call `print` on an instance of the `A` class, it behaves as expected. If we then set the `print` method to point at a new function, it tells us something different:

```
my class is A
my class is not A
```

It is also possible to replace methods on classes, instead of objects, although in that case we have to add the `self` argument to the parameter list. This will change the method for all instances of that object, even ones that have already been instantiated.

Obviously, replacing methods like this can be very dangerous and confusing to maintain. Somebody reading the code will see that a method has been called, and look up that method on the original class. But the method on the original class is not the one that was called. Figuring out what really happened can become a very tricky debugging session.

It does have its uses though. Often, replacing or adding methods at run time (called **monkey-patching**) is used in automated testing. If testing a client-server application, we may not want to actually connect to the server when testing the client; that may result in accidental transfers of funds or embarrassing test e-mails being sent to real people. Instead, we can set up our test code to replace some of the key methods on the object that sends requests to the server, so it only records that the methods have been called.

Monkey-patching can also be used to fix bugs or add features in third-party code that we are interacting with and does not behave quite the way we need it to. It should, however, be applied sparingly, it's almost always a "messy hack". Sometimes, though, it is the only way to adapt an existing library to suit our needs.

Callable objects

Since functions are just objects that happen to respond to the call syntax, we start to wonder if it's possible to write objects that can be called yet aren't real functions. Yes, of course!

Any object can be turned into a callable, as easily as giving it a `__call__` method that accepts the required arguments. Let's make our `Repeater` class from the timer example a little easier to use by making it a callable:

```
class Repeater:
    def __init__(self):
        self.count = 0
    def __call__(self, timer):
        format_time("{now}: repeat {0}", self.count)
        self.count += 1
        timer.call_after(5, self)

timer = Timer()
timer.call_after(5, Repeater())
format_time("{now}: Starting")
timer.run()
```

This example isn't much different from the earlier class; all we did was change the name of the `repeater` function to `__call__` and pass the object itself as a callable. Note that when we make the `call_after` call, we pass the argument `Repeater()`. Those two parentheses are creating a new instance of the class, they are not explicitly calling the class. That happens later, inside the timer. If we want to execute the `__call__` method on a newly instantiated object, we'd use a rather odd syntax: `Repeater()()`. The first set of parentheses constructs the object; the second set executes the `__call__` method.

Case study

To tie together some of the principles presented in this chapter, let's build a mailing list manager. The manager will keep track of e-mail addresses categorized into named groups. When it's time to send a message, we can pick a group and send the message to all e-mail addresses assigned to that group.

Now, before we start working on this project, we ought to have a safe way to test it, without sending e-mails to a bunch of real people. Luckily, Python has our back here; like the test HTTP server, it has a built in **Simple Mail Transfer Protocol** (**SMTP**) server that we can instruct to capture any messages we send without actually sending them. We can run the server with the following command:

```
python -m smtpd -n -c DebuggingServer localhost:1025
```

Running this command at a command prompt will start an SMTP server running on port 1025 on the local machine. But we've instructed it to use the `DebuggingServer` class (it comes with the built-in SMTP module), which, instead of sending mails to the intended recipients, simply prints them on the terminal screen as it receives them. Neat, eh?

Now, before writing our mailing list, let's write some code that actually sends mail. Of course, Python supports this in the standard library too, but it's a bit of an odd interface, so we'll write a new function to wrap it all cleanly:

```
import smtplib
from email.mime.text import MIMEText

def send_email(subject, message, from_addr, *to_addrs,
        host="localhost", port=1025, **headers):

    email = MIMEText(message)
    email['Subject'] = subject
    email['From'] = from_addr
    for header, value in headers.items():
        email[header] = value

    sender = smtplib.SMTP(host, port)
    for addr in to_addrs:
        del email['To']
        email['To'] = addr
        sender.sendmail(from_addr, addr, email.as_string())
    sender.quit()
```

We won't cover the code inside this method too thoroughly; the documentation in the standard library can give you all the information you need to use the `smtplib` and `email` modules effectively.

We've used both variable argument and keyword argument syntax in the function call; any unknown arguments are mapped to extra addresses to send to; any extra keyword arguments are mapped to e-mail headers.

The headers passed into the function represent auxiliary headers that can be attached to a method. Such headers might include Reply-To, Return-Path, or X-pretty-much-anything. Can you see a problem here?

Any valid identifier in Python cannot include the `-` character. In general, that character represents subtraction. So it's not possible to call a function with `Reply-To = my@email.com`. Perhaps we were too eager to use keyword arguments because they are a new tool we just learned this chapter?

We'll have to change the argument to a normal dictionary; this will work because any string can be used as a key in a dictionary. By default, we'd want this dictionary to be empty, but we can't make the default parameter an empty dictionary. No, we'll have to make the default argument `None`, and then set up the dictionary at the beginning of the method:

```
def send_email(subject, message, from_addr, *to_addrs,
        host="localhost", port=1025, headers=None):

    headers = {} if headers is None else headers
```

If we have our debugging SMTP server running in one terminal, we can test this code in a Python interpreter:

```
>>> send_email("A model subject", "The message contents",
 "from@example.com", "to1@example.com", "to2@example.com")
```

Then if we check the output from the debugging SMTP server, we get the following:

```
---------- MESSAGE FOLLOWS ----------
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: A model subject
From: from@example.com
To: to1@example.com
X-Peer: 127.0.0.1

The message contents
```

```
------------ END MESSAGE ------------
---------- MESSAGE FOLLOWS ----------
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: A model subject
From: from@example.com
To: to2@example.com
X-Peer: 127.0.0.1

The message contents
------------ END MESSAGE ------------
```

Excellent, it has "sent" our e-mail to the two correct addresses with subject and message contents included.

Now that we can send messages, let's work on the e-mail group management system. We'll need an object that somehow matches e-mail addresses with the groups they are in. Since this is a many-to-many relationship (any one e-mail address can be in multiple groups, any one group can be associated with multiple e-mail addresses), none of the data structures we've studied seem quite ideal. We could try a dictionary of group-names matched to a list of associated e-mail addresses, but that would duplicate e-mail addresses. We could also try a dictionary of e-mail addresses matched to groups, resulting in a duplication of groups. Neither seems optimal. Let's try this latter version, even though intuition tells me the groups to e-mail address solution would be more straightforward.

Since the values in our dictionary will always be collections of **unique** e-mail addresses, we should probably store them in a `set`. We can use `defaultdict` to ensure there is always a `set` available for each key:

```
from collections import defaultdict
class MailingList:
    '''Manage groups of e-mail addresses for sending e-mails.'''
    def __init__(self):
        self.email_map = defaultdict(set)

    def add_to_group(self, email, group):
        self.email_map[email].add(group)
```

Now let's add a method that allows us to collect all the e-mail addresses in one or more groups. We can use a set comprehension to take care of this easily:

```
    def emails_in_groups(self, *groups):
        groups = set(groups)
        return {e for (e, g) in self.email_map.items()
                if g & groups}
```

OK, that set comprehension needs explaining, doesn't it? First look at what we're iterating over: `self.email_map.items()`. That method, of course, returns a tuple of key-value pairs for each item in the dictionary. The values are sets of strings representing the groups. We split these into two variables named `e` and `g`, short for e-mail and groups. We only return the key (the e-mail address) for each item though, since the desired output is a set of e-mail addresses.

The only thing left that may not make sense is the condition clause. This clause simply intersects the `groups` set with the set of groups associated with the e-mails. If the result is non-empty, the e-mail gets added, otherwise, it is discarded. The `g & groups` syntax is a shortcut for `g.intersection(groups)`; the `set` class does this by implementing the special method `__and__` to call `intersection`.

Now, with these building blocks, we can trivially add a method to our `MailingList` class that sends e-mail to specific groups:

```
    def send_mailing(self, subject, message, from_addr,
            *groups, **kwargs):
        emails = self.emails_in_groups(*groups)
        send_email(subject, message, from_addr,
                *emails, **kwargs)
```

This function stresses on variable arguments. As input, it takes a list of groups as variable arguments, and optional keyword arguments as a dictionary. It doesn't care about the keyword arguments at all; it simply passes those arguments on to the `send_email` function we defined earlier. It then gets the list of e-mails for the specified groups, and passes those as variable arguments into `send_email`.

The program can be tested by ensuring the SMTP debugging server is running in one command prompt, and, in a second prompt, load the code using:

```
>>> python -i mailing_list.py
```

Create a `MailingList` object with:

```
>>> m = MailingList()
```

Then create a few fake e-mail addresses and groups, along the lines of:

```
>>> m.add_to_group("friend1@example.com", "friends")
>>> m.add_to_group("friend2@example.com", "friends")
>>> m.add_to_group("family1@example.com", "family")
>>> m.add_to_group("pro1@example.com", "professional")
```

Finally, use a command like this to send e-mails to specific groups:

```
>>> m.send_mailing("A Party",
"Friends and family only: a party", "me@example.com", "friends",
"family", headers={"Reply-To": "me2@example.com"})
```

E-mails to each of the addresses in the specified groups should show up in the console on the SMTP server.

Exercises

If you don't use comprehensions in your daily coding very often, the first thing you should do is search through some existing code and find some `for` loops. See if any of them can be trivially converted to a generator expression or a list, set, or dictionary comprehension.

Test the claim that list comprehensions are faster than `for` loops. This can be done with the built-in `timeit` module. Use the help documentation for the `timeit.timeit` function to find out how to use it. Basically, write two functions that do the same thing, one using a list comprehension, and one using a `for` loop. Pass each function into `timeit.timeit`, and compare the results. If you're feeling adventurous, compare generators and generator expressions as well. Testing code using `timeit` can become addictive, so bear in mind that code does not need to be hyper-fast unless it's being executed an immense number of times, such as on a huge input list or log file.

Try writing the case study using groups as dictionary keys and lists of e-mail addresses as the values. You'll likely be surprised at how little needs to be changed. If you're interested, try reworking it to accept first and last names as well as e-mail addresses. Then allow customizing e-mail messages (use `str.format`) to have the person's first or last name in each message.

Except the `send_mailing` method itself, the `MailingList` object is really quite generic. Consider what needs to be done to make it perform any generic activity on each e-mail address, instead of just sending mail. Hint: callback functions will be very useful.

Play around with generator functions. Start with basic iterators that require multiple values (mathematical sequences are canonical examples; the Fibonacci sequence is overused if you can't think of anything better). Try some more advanced generators that do things like take multiple input lists and somehow yield values that merge them. Generators can also be used on files; can you write a simple generator that shows those lines that are identical in two files?

Summary

We covered several very different topics in this chapter. Each represented an important non-object-oriented feature that is popular in Python. Just because we can use object-oriented principles does not always mean we should!

However, we also saw that "the Python way" often just provides a shortcut to traditional object-oriented syntax. Knowing the object-oriented principles underlying these tools allows us to use them effectively in our own classes.

We covered:

- Built-in functions
- Comprehensions and generators
- Function arguments, variable arguments, and keyword arguments
- Callback functions and callable objects

In the next chapter, we're going to study design patterns; building blocks that object-oriented programmers use to create maintainable applications. In many cases, we'll see that, as in this chapter, Python provides syntax for popular design patterns that we can use instead.

Where to buy this book

You can buy Python 3 Object Oriented Programming from the Packt Publishing website:
`https://www.packtpub.com/python-3-object-oriented-programming/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

[PACKT] PUBLISHING open source community experience distilled

www.PacktPub.com

For More Information:
www.PacktPub.com/python-3-object-oriented-programming/book